Chapter 14

Statistical Analysis

This chapter describes the CMSSL statistical analysis routines. Currently included are two histogramming operations that summarize a specified CM array by the number of occurrences of each value or range of values. The following three routines are provided:

histogram	Histograms all values in a data set.
histogram_range **histogram_range_cm**	Histogram designated ranges of values within a data set.

Histograms provide a statistical mechanism for simplifying data. They are generally used in applications that need to display or extract summary information. For particularly large data sets, **histogram_range** and **histogram_range_cm** facilitate breaking data down into subranges, perhaps as a preliminary step before doing more detailed analysis of interesting areas.

In particular, histograms have many applications in image analysis and computer vision. For example, a technique known as histogram equalization computes a histogram of pixel intensity values in an image and uses it to rescale the original picture.

The CMSSL histogram operations treat the elements of an array as a series of *bins*. In the **histogram** and **histogram_range** routines, the bins are represented by a front-end array; in the **histogram_range_cm** routine, they are represented by a CM array. In each bin, a tally of values or value ranges is stored. The number of histogram bins varies widely with the application, from a dozen tallies on a large process or a few dozen markers on a probability distribution to a few hundred intensity values in an image or a few thousand instruction codes in a performance analysis.

Copyright © 1994 Thinking Machines Corporation

14.1 How to Histogram

To decide whether to use a simple or a ranged histogram, consider the number of bits needed to represent the source values that are to be analyzed. For a simple histogram, one front-end array element (or *bin*) is required for each possible source value. Since 2^{length} possible values can be represented in *length* bits, a simple histogram requires 2^{length} front-end bins. From this, we see that 8-bit values can be histogrammed into 2^8 or 256 bins—a manageable number. However, a simple histogram of 16-bit values would require 2^{16} or 65,536 bins—which is probably too many for useful analysis.

A range histogram uses one bin for each range of source values. Use a range histogram to analyze source arrays that include large values. Determining the number of bins to use and the range to assign each bin is easily done using exponent arithmetic. Suppose for example we have 16-bit source values. Since $2^{16} = 2^8 \times 2^8$, we could use $2^8 = 256$ bins and tally a range of 2^8 values in each. When making these calculations, don't forget to account also for two tail bins—one for either end of the value range. (See the **histogram_range** or **histogram_range_cm** man page for more on tail bins.)

As a concrete illustration of histogramming techniques, consider summarizing the pixel information of a framebuffer image so that each bin reflects the number of pixels with a given intensity.

Suppose the intensity is encoded in a CM array of 8-bit elements, as for black and white images. The range of possible values is small, so a simple histogram can be used. The front-end array should have one element (or bin) for each possible source value, or $2^8 = 256$ elements. When we invoke the histogram routine

```
CALL histogram (fe_array, source_array, sbit_len, ier)
```

each bin tallies the number of pixels for one discrete intensity level. Note: A histogram call on 8-bit elements may be simulated by using a *length* argument of 8 to histogram the low-order 8 bits of each 32-bit element.

If, however, the pixel intensity is encoded in a CM array of 24-bit elements, as it is for RGB color images, then a front-end array of 2^{24} = 16,777,216 elements would be too large. (Also, an array with 2^{24} 32-bit words would occupy over 67 megabytes.) Fortunately, there are two alternatives:

1. Call a simple histogram on each 8-bit color subfield separately, as with the black and white case described above.

Copyright © 1994 Thinking Machines Corporation

2. Use a range histogram as a microscope to close in on the interesting parts of the data, as described below.

To use a range histogram for this problem, try to factor the total 2^{24} range into manageable chunks. In particular, since $2^{24} = 2^{16} \times 2^8$, we could put 2^{16} values in each of 2^8 middle range bins and leave the tail bins empty. Following this course, we can obtain an overview of all the data by invoking the range histogram routine:

```
CALL historam_range (fe_array, source_array, 258, 0, 65536 )
```

In this call, we specify the number of bins as 258 (256 + 2 tail bins = 258), the range minimum as 0, and the range within each bin as 2^{16} = 65536.

Copyright © 1994 Thinking Machines Corporation

Histogram

Increments each element of the specified front-end array by the number of times its index equals the value of an element in the specified source array.

For particularly large ranges of source data, consider using the **histogram_range** or **histogram_range_cm** routine.

SYNTAX

histogram (*s, A, length, ier*)

ARGUMENTS

s	Front-end array of 32-bit integers for storing the histogram. The number of elements in this array should be at least 2^{length}.
A	Source CM array of type integer (32-bit or 64-bit).
length	Length in bits of the *A* values, or the number of bits of each *A* value that are to be considered by the histogram routine. The specified *length* should be no greater than *n* such that an *s* array of 2^n elements can fit in partition manager memory.
ier	Error code. Scalar integer. Set to 0 if the routine succeeds.

DESCRIPTION

A tally is made of the number of times any particular integer *i* occurs as a value of *A*. This tally is added to the value of the corresponding histogram bin, *s*[*i*]. For example, if *s* has been initialized to zero, and if *s* contains five elements that have the value 3, then *s*(3) receives the value 5.

The histogram thus records the distribution of values within one or more source arrays (one per call). Note that histogram bins (that is, elements in *s*) are not pre-cleared automatically. This allows the successive collection of data from many arrays into one histogram.

The *length* value is the number of bits of each *A* value that are to be considered by the histogram routine. For example, if *A* contains 32-bit integers, a *length* value of 8

Copyright © 1994 Thinking Machines Corporation

causes the values represented by the low 8 bits of each source value to be tallied. High-order bits are ignored.

NOTES

No Error Checking. No special error checking is done. Thus, there may be unpredictable damage if there are not enough destination bins to hold all values up through the maximum value of the source data.

Front-End Array Size. To avoid error, the correct number of front-end array elements is 2^{length}, where *length* is the number of bits used to represent each ***A*** value.

Performance. With the current implementation of this histogram routine, optimum performance is obtained using large source arrays and a small number of bins.

Zero-Based Array Indexing. Zero values are tallied in the first bin. For this reason, CM Fortran users might want to declare *s* to be zero-based.

EXAMPLES

Sample code that uses the histogram routines can be found on-line in the subdirectory

```
histogram/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

Range Histogram

The range histogram routines increment each element of an array by the number of times a source value falls into the subrange associated with that element. The array that is incremented is a front-end array for the **histogram_range** routine, and is a CM array for the **histogram_range_cm** routine.

These routines are particularly useful for obtaining a statistical summary of large data sets, for which unranged histogram information is too unwieldy to analyze. To histogram small data sets, consider using the **histogram** routine.

SYNTAX

histogram_range (*s, **A**, n, min, range, ier*)

histogram_range_cm (*s, **A**, n, min, range, ier*)

ARGUMENTS

s	Front-end array (for **histogram_range**) or CM array (for **histogram_range_cm**) of 32-bit integers. Stores the histogram. The number of elements in this array must be greater than or equal to *n*.
A	Source CM array containing either real or integer (32-bit or 64-bit) values.
n	Scalar 32-bit integer specifying the number of destination bins, including 2 bins for tail values.
min	Scalar variable of the same type as ***A***. Specifies the minimum value of the elements of ***A*** placed in the second bin.
range	Scalar variable of the same type as ***A***. Specifies the size of the subrange associated with all but the first and last bins. Must be greater than 0.
ier	Error code. Scalar integer variable. Set to 0 on successful return; set to -1 if overflow occurs (see the Notes section below) or if *range* ≤ 0.

Copyright © 1994 Thinking Machines Corporation

DESCRIPTION

These routines treat each element of *s* as a bin associated with a range of ***A*** values. To each bin they assign an integer value equal to the bin's initial value plus the number of values within ***A*** that fall into the bin's range.

There must be at least *n* elements in *s*. The range histogram uses the first bin, *s*[1], and the last bin, *s*[*n*], to tally ***A*** values below and above the range under inspection. The middle elements are used to tally ***A*** values within subranges that span successive increments of *range*. The *min* and *range* parameters determine the range of values contributing to each element of the histogram, as follows:

- The first bin, *s*[1], is incremented by the tally of all ***A*** values *j* for which $j <$ *min*.
- Each middle bin, *s*[*i*] ($1 < i < n$), is incremented by the tally of ***A*** values *j* for which
  $j \geq (min + (i - 2) \times range)$
  and
  $j < (min + (i - 1) \times range)$.
- The last bin, *s*[*n*], is incremented by the tally of ***A*** values *j* for which $j \geq (min + (n - 2) \times range)$.

The histogram thus records the distribution of values within one or more source arrays (one per call). Note that histogram bins are not pre-cleared automatically. This allows the successive collection of data from many CM arrays into one histogram.

NOTES

Overflow. If you supply *min*, *range*, and *n* values that result in an overflow, *ier* is set to −1. In the case of integer data, an overflow occurs if the lowest element in the last bin, bin *s*[*n*], exceeds (*max_int* + 1), where *max_int* is the maximum representable integer. An equivalent statement is that overflow occurs if the highest value in bin *s*[*n*−1] exceeds *max_int*. Using the formulas in the Description section, we see that in the case of single-precision integer ($max_int = 2^{31} - 1$), overflow occurs if

$$min + [(n - 2) \times range] > 2^{31} \qquad \text{(overflow, single-precision integer)}$$

Copyright © 1994 Thinking Machines Corporation

For example, in the single-precision case, if

$min = 2^{31} - 15$
$range = 1$
$n = 18$

then $min + [(n - 2) \times range] = 2^{31} + 1$, and overflow occurs; whereas if

$min = 2^{31} - 16$
$range = 1$
$n = 18$

then $min + [(n - 2) \times range] = 2^{31}$, and overflow is avoided.

Performance. To obtain optimal performance with **histogram_range_cm**, you must use the vector-unit execution model; ***A*** must be a 32-bit integer array; *range* must be a power of 2; and *n* must be 2 more than a power of 2. If you satisfy these conditions, **histogram_range_cm** is much faster than **histogram_range**, especially as the number of bins increases.

If the conditions described above are not satisfied, with the current implementation of the histogram range routines, best performance is obtained with large source arrays and a small number of bins.

EXAMPLES

Sample code that uses the histogram routines can be found on-line in the subdirectory

```
histogram/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation